Heterogeneous student grouping by hybrid particle swarm optimization

Paul Maguire
Athabasca University
paul@pmetal.ca

Rajan Kapila
Athabasca University
rajankapila989@hotmail.com

Abstract: Students described by a list of personality attribute scores must be grouped to satisfy a constrained optimization problem. A good grouping assembles four students with two outliers surrounding a coherent core. A "Goodness of Heterogeneity" score for each candidate solution was maximized by a hybrid particle swarm optimization algorithm using various different combinations of parameter values. By varying the control parameters, we were able to determine which parameters contribute to algorithmic performance and discover several satisfactory groupings.

Introduction

The problem addressed by this paper is the implementation of a Computational Intelligence (CI) algorithm of our choice to search for good solutions to a student grouping problem. At first glance, the reader might suspect the student grouping problem specification describes a clustering problem. The success of artificial neural nets with categorization problems [1] has early appeal. However, this grouping is a bit of a red herring. The goodness of heterogeneity alters the perceived problem by requiring differences between group members. The student grouping is revealed as an optimization problem.

The objective function is clearly articulated. A student cohort must be divided into groups, subject to several constraints. Each student belongs to exactly one group; each group has exactly four students; and at least one student pair must be separated by a Euclidean distance (in attribute scores) of at least 2. In addition, each group must have a goodness of heterogeneity (GH) score greater than 0.5. The objective function maximizes the sum of the group GH scores. Anecdotal evidence suggests that, for this problem, "very good solutions have a heterogeneity of over 600" [2].

From the Computational Intelligence family, we have already investigated genetic algorithms (GAs). Research suggests that particle swarms can converge quickly and require a small set of parameters [3]. Optimal settings have been suggested for some of the parameters. Personal interest, an appealing simplicity and time constraints combined to push us to investigate Particle Swarm Optimization (PSO) for this project. The reported speed of convergence and minimal parameterization both suggest advances over GA performance and tuning [3].

Particle Swarm Optimization algorithm

Particle Swarm Optimization (PSO) was inspired by flocking birds and fish [4]. Under PSO, a population of candidate solutions iterates towards an optimal problem solution. Each particle within a swarm is influenced by its own best performance and the best performance of a neighbouring particle. PSO has low memory requirements, as each particle must know only its current position, its best position encountered so far, and velocity information. Under the canonical PSO, each particle uses its velocity to adjust its position in each generation. The velocity changes each generation according to the particle's current position, its best ever position, and some neighbour particle's best ever position.

The search space dimensionality varies with the objective function. Each particle's position is expressed as a point within the D-dimensional search space. The *i*th particle has position $x_i = (x_{i1}, x_{i2}, \ldots, x_{iD})$; at time (or generational

step) t, the position of the ith particle is x_i^t. The velocity of the ith particle at time t is $v_i^t = (v_{i1}^t, v_{i2}^t, \ldots, v_{iD}^t)$. Each particle's generational velocity and position change can be expressed algebraically:

$$v^{t+t} = w \bullet v_i^t + c_1 \bullet r_1 \bullet (x_i^{best} - x_i^t) + c_2 \bullet r_2 \bullet (n^{best} - x_i^t)$$

$$x_i^{t+1} = x_i^t + v^{t+1}$$

In the equations above, w is an inertial weight that has the effect of altering the environmental viscosity [3]; c_1 and c_2 are user defined acceleration constants (also referred to as learning rates); and r_1 and r_2 are uniformly distributed random numbers; n^{best} is the best known position within the neighbourhood; and x_i^{best} is the ith particle's best position to date. Defining a neighbourhood can be computationally expensive. In the worst case, for a fully informed particle swarm, each pair of particles must be compared, which is an $O(n^2)$ task. Different topologies have been investigated [5]. The general approach is to track only a swarm's overall best particle – i.e., the neighbourhood is the entire swarm.

The quality of a position is gauged using a fitness function, which is an expression of an optimization problem's objective function.

The student grouping problem

A solution to the student grouping problem (SGP) is an ordering of students. Therefore, we looked for PSO solutions to permutation problems. We first looked for algorithms that deal with the Traveling Salesman Problem (TSP). TSP is an optimization problem that seeks a minimal Hamiltonian path in a weighted graph. SGP and TSP problems are not perfectly analogous as TSP edge weights define a relationship with a particle that has no direct comparison in the SGP. However, the commonality is sufficient as we seek only to borrow an approach that will alter particle positions in a way that will produce a viable solution. Valid solutions are permutations that include each student exactly once.

While Ant Colony Optimization (ACO) has been acknowledged as effective for solving TSP [6][7][8], PSO has not gained similar recognition.

Our approach

Various researchers have investigated using PSO to address TSP. An early work approached the problem by defining swap sequence and a swap operator that, directed by a personal or neighbourhood best position, relocates with a random probability some part of a particle's dimensional values [9]. Another swap operator, repeatedly exchanges pairs of a particle's dimensions, with repetition frequency dependent upon distance between particles as measured by fitness scores [10]. Another approach borrows from ACO, augmenting particle memory with something akin to pheromone trails and allowing a particle's velocity to be influenced by any of several previous personal bests [6]. Another investigates connecting PSO's continuous search space with TSP's discrete solution space as a space transformation mapping problem [7]. One more approach builds effectively on the swap operator concept by creating a permutation concept, which they support with several new operators and definitions [11].

Of these published approaches, several leave elements less than fully articulated.

Another paper presents Improved Hybrid Discrete Particle Swarm Optimization (IHDPSO), which redefines the canonical velocity update procedure [12]. IHDPSO includes a crossover reminiscent of GA, which is accompanied by inject/reverse mutation and adaptive noise operations. The crossover operation most closely resembles Order Crossover [13].

Reasoning that a seeming hybridization of PSO with GA might provide positive results, we chose to follow this IHDPSO methodology. The originating author (OA) claims positive results, but fails to fully articulate the methods. We adapted what we could of his algorithm.

With an incomplete implementation of IHDPSO and recognizing PSO's difficulties in solving discrete problems, we decided to further hybridize our solution. In addition to IHDPSO, we adjusted the particle population into multiple swarms. Our model for this is Dynamic Multi-Swarm Particle Swarm Optimizer (DMSPSO) [14]. We describe this algorithm below.

Our hybridization of these two solutions is an attempt to address PSO's underperformance (relative to ACO) in discrete problems. Updating PSO with GA influences in a multi-swarm configuration has been tried before [15]. Our effort differs from previous work most significantly in the IHDPSO crossover approach. Where IHDPSO more or less follows Order Crossover, another approach creates offspring particles arithmetically: $child_1 = r_1 \bullet parent_1 + (1 - r_1) \bullet parent_2$, and the new child's velocity vectors are normalized sums of the parents' velocity vectors [15]. That arithmetic approach's modest results leave room for our attempt to achieve better results with our different strategy.

Improved Hybrid Discrete Particle Swarm Optimization

In essence, the IHDPSO procedure to update particle velocity follows the canonical PSO pattern. The crossover operator starts with the current position, and applies components influenced by the particle's prior best position and the swarm's best position to date. Crossover is expressed (by the algorithm's creator) as

$$x_i^{t+1} = x_i^t \oplus ((c_1 \bullet r_1)\, x^{best}) \oplus ((c_2 \bullet r_2)\, n^{best})$$

where c_1 and c_2 are parameterized values, $r1$ and $r2$ are random over [1..D] [1] (D is the problem's dimensionality, which is the same as the particle "size"), x^{best} is the particle's best position so far, n^{best} is the best result found to date in the neighbourhood. The binary crossover operator $\oplus$ is more clearly understood as

$$x' = x \oplus ((c \bullet r)\, y)$$

which means the crossover operator updates particle x by extracting a "particle substring" from y. The substring starting point k is a random dimension within the particle, and substring length m is $(c \cdot r)$. Remembering this is a permutation, x' is filled with the remaining dimensional values from x that are not already present in the substring, from the beginning and preserving their order (see figure 1).

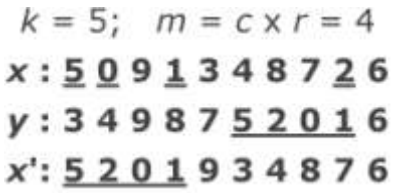

Figure 1 - IHDPSO crossover

In the original IHDPSO, crossover is followed by the inject/reverse operation. This is in fact two operators. At a random coin flip, either an inject or a reverse operator can be applied. As the algorithm's creator omitted detail on the inject operator, we apply only the reverse operator. The reverse is quite simple. Random points within the particle are selected, and the order between them is reversed. For example, with random start position = 2 and end position = 5, {3,**1,5,4,2**,6} becomes {3,**2,4,5,1**,6}.

IHDPSO's third step in particle velocity change introduces noise and is intended to maintain population diversity. Again, the creator omits necessary details, and we omitted this step.

In summary, we implemented 1.5 of the 3 steps in the IHDPSO algorithm for modifying particle velocity.

Dynamic Multi-Swarm Particle Swarm Optimizer

We chose to augment our approach to the problem with a feature we hoped would be unique, and that might provide better results than had other PSO approaches. On top of the modified velocity adjustment, we added a layer that divides the global particle population into several smaller, equally-sized swarms, following Dynamic Multi-Swarm Particle Swarm Optimizer (DMSPSO) [14]. For several benchmark functions, DMSPSO achieved results as good as or better than seven other PSO approaches for almost every function (second best behind fitness-distance ratio PSO against Rosenbrock's Function).

DMSPSO requires the following parameters:

- m is the size of each swarm;

[1] The IHDPSO author does not specify a range or meaning for r_1 and r_2. We chose to use particle size (D) to define the range.

- n is the number of swarms;
- the particles are randomly reassigned to a different swarm every R generations, where R is the regrouping period;
- *Max_gen* is the maximum number of generations to iterate through and is used as a termination criterion;
- particles initially move through only their own swarm, influenced by the swarm's n^{best} for a portion of the *Max_gen* generations. *Local_Trial_Pct* is the threshold at which the local movement terminates, and all particles are influenced instead by the global best particle to date from all swarms (g^{best}). In essence, after *Local_trial_Pct* • *Max_gen* generations, all particles resolve into a single global swarm.

The DMSPSO algorithm follows these steps:

1. Initialize $m \bullet n$ particles with random position and velocity
2. Randomly divide the particles into n swarms of size m
3. For $i := 1$ .. *Local_Trial_Pct* • *Max_gen*:

   Update each particle using its swarm's best n^{best}

   If $mod(i, R) = 0$, randomly reassign particles to different swarms
4. For $i :=$ *Local_Trial_Pct* • *Max_gen* .. *Max_gen*:

   Update each particle using population's best g^{best}

DMSPSO coexists quite comfortably with IHDPSO. Each of the DMSPSO swarms operates independently, and according to the direction of our modified IHDPSO algorithm. In the traditional PSO style, IHDPSO moves a particle through the search space towards its own personal best position and that of a neighbourhood best particle (n^{best}). The n^{best} is established by DMSPSO. Otherwise, the two algorithms are essentially invisible to each other.

Results

Experimental approach

Our approach to solving the problem was to run our program over several small generation limits to identify better parameter settings. Default parameters for these trials were population size 30, generations 200, local trial percentage 90, regrouping period 5.

Optimal values are those which generate the highest scores from the fitness function (FF).

We are sensitive to the possibility that parameters might share relationships that affect algorithmic performance. Such eventualities are extremely difficult to track down and would require much time and effort, for reasons similar to observations reported for GA [16].

Parameter setting

c_1 and c_2. The IHDPSO crossover uses parameters c_1 and c_2 to influence the velocity modification. We ran several small trials of 200 generations against different combinations for c_1 and c_2. Parameters were set to default values as specified above, with swarm size = 3. Results shown in table 1 show the averaged results of 10 trials for each parameter combination, with all other parameters held constant. Trials indicate best results for c_1 and c_2 as 0.8 and 1.0, respectively.

c_1 \ c_2	0.2	0.4	0.8	0.9	1.0
0.2	239.18	239.79			
0.4	238.48	240.83			
0.6	239.73	241.73	258.83	258.36	260.06
0.8	240.06	241.90	259.62	260.52	**269.47**
1.0			262.98	262.52	264.05

Table 1 - Seeking optimal c1/c2

Swarm size. Investigation into optimal swarm size was handled in a similar fashion. Several 200-generation runs were held with results averaged over 10 trials for each set of parameters, with c_1 and c_2 now set at 0.8 and 1.0, respectively. The results shown in table 2 show no significant impact for changing swarm size. The original DMSPSO work tested swarm sizes of 2, 3 and 5, and found that swarm size of either 3 or 5 might provide better results, depending on the problem. As swarm size 5 was marginally (though not significantly) preferable in our testing as well, we chose to use that setting for our algorithm.

Swarm size	2	3	5	6	10	15
Avg FF score	262.05	262.90	264.30	265.63	264.59	262.70

Table 2 - Seeking optimal swarm size

Population size. We tested several different population sizes, all with swarm size = 5. As above, we ran each parameter set through 10 trials over 200 generations each, and show the averaged results in table 3 and graphically in figure 3. It is clear the average score increases with population size. This follows logically as an increased population means increased testing. However, the increased population makes increased demands on processing resources, and the jobs take longer to complete (see figure 2). We compromised on a population size of 100 particles.

Pop	10	50	100	200	500	750	1000[2]	1250	1500
Avg FF score	242.68	259.60	272.53	320.79	343.73	391.95	395.78	404.56	410.48

Table 3 - Seeking optimal population size

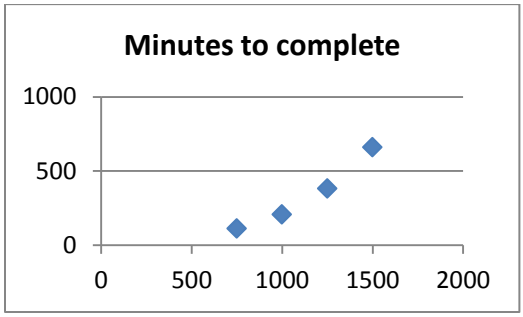

Figure 2 - Minutes to complete 10 trials by population size (x-axis)

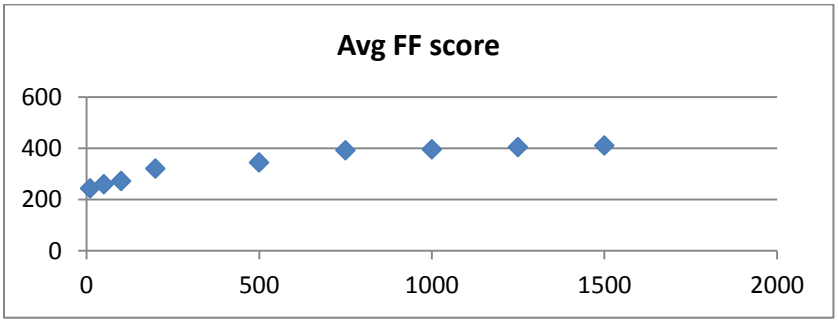

Figure 3- Average FF score by population size (x-axis)

Limited testing for local trial percentage and regrouping period showed little variation. Consequently, we chose to use the original authors' values for these (90% and 5, respectively).

Problem solution with tuned parameters

Having established our tuned parameter settings, we undertook a larger scale search for good solutions. We ran 10 trials with parameters set as established above, for 100,000 generations each. After three days running on a 2.66

[2] Results shown for population size 1000 are the average of 20 trials, rather than 10 trials.

GHz Intel core i7 processor with 8GB RAM, the program completed. As shown in Figure 4, our algorithm explores effectively and identifies good solutions. However, it takes about 28,000 generations to reach a 600 average fitness score. This is significantly slower than our GA, which reached a higher maximum in around 20,000 generations.

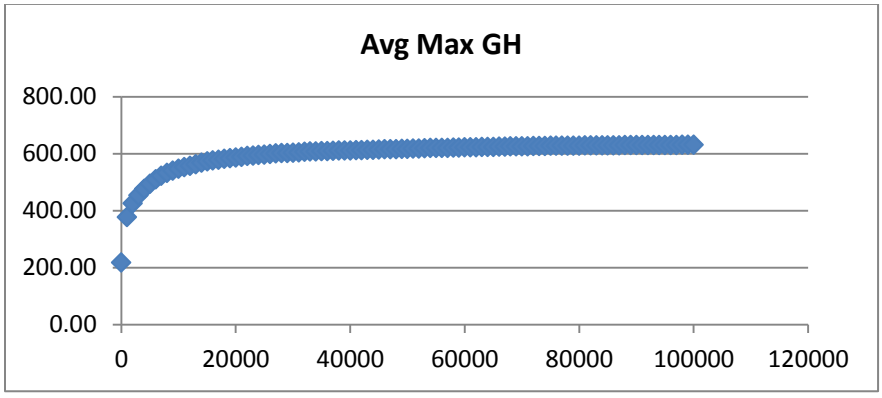

Figure 4 - Average maximum GH against generation count (x-axis)

This result was achieved with $c_1 = 0.8$, $c_2 = 1.0$, $R = 5$, population = 100, regrouping after 5 generations, and swarm size = 5.

How to run our program

The program is launched with the following command:

```
java –jar comp658-psov1.jar
```

A file named `runs-pso-trials.txt` must be found in the directory with the executable `.jar` file. Records in this input text file include a set of parameter and value pairs separated with " : ". These pairs are comma separated. For example, one line might be:

```
populationSize : 100, swarmCount : 20, regroupPeriod : 5, crosspt1 : 0.8, crosspt2
: 1.0, maxGenerations : 2000, dataPointFrequency : 50, localTrialPct : 0.9, trials
: 10
```

This input file can be several lines long, resulting in one program run for each line with the different parameters. Note the `trials` parameter will initiate duplicate executions with the same parameter set. In other words, three lines in this input file, each containing `trials : 10`, will execute our program 30 times.

The parameters above control operation as follows.

- The `populationSize` field is an integer that determines how many particles will be created.
- The `swarmCount` field requires an integer, and defines the number of swarms that will be created. It is important that `populationSize` is evenly divisible by `swarmCount`. If all swarms are not the same size, the program will crash.
- The `regroupPeriod` field is an integer that defines the number of generations between swarm regroupings.
- The `crosspt1` field is a real number in the range [0.0 – 1.0] that defines the probability that the particle's best score will be considered in calculating its next velocity.
- The `crosspt2` field is a real number in the range [0.0 – 1.0] that defines the probability that the neighbourhood's (i.e., local swarm) best score will be considered in calculating the particle's next velocity.
- The `dataPointFrequency` integer field is for reporting purposes only. It defines how frequently a snapshot is written to the output data file.
- The `localTrialPct` is a real-valued field that identifies when the program switches from evaluating a particle's next velocity locally (within its swarm), or globally (across all swarms). A value of 0.9 means

that 90% of the generations will calculate velocity locally before the final 10% of generations use the global best particle for all velocity calculations. (In other words, this defines the point at which all swarms are merged into one.)
- The `trials` parameter requires an integer that identifies how many times to run this parameter control set.
- The `maxGenerations` parameter defines how many iterations the algorithm will cycle through for each trial.

A second input file named `input.txt` must be present in the same directory as the program `.jar` file. The input data file expects a file where each line contains a student number and seven attribute scores, all comma separated. This input file is expected to include 512 student records. For example, student number 16 might look like this:

```
16,2,1,2,2,1,1,3
```

Two output files are generated. Snapshots are captured every `dataPointFrequency` generations and are written to `output_<datestamp>.csv`. Summarized scores across multiple trials of a single parameter set are written to `trials_<datestamp>.txt`.

Comparison: Genetic Algorithm VS. Particle Swarm Optimization

The main difference between the genetic algorithm and particle swarm optimization algorithm is the way the search space is searched. The genetic algorithm uses a "survival of the fittest" approach by promoting the best solutions in each generation, along with breeding the elite solutions and adding randomness with mutation. The particle swarm optimization algorithm explores the search space by moving solutions towards the best solutions found so far. The best solutions are not promoted or kept each generation but each solution in the swarm partially moves toward the best solution each time the swarm moves. Although similar in how they act, the genetic algorithm pushes the population to a higher average score as the best solutions are promoted whereas the particle swarm optimization algorithm's population is much more random and best solutions do not persist over time.

Our modifications to the canonical GA and PSO algorithms transformed both of these continuous problem solvers into discrete permutation machines. In each case, we altered their key operators. GA's crossover and PSO's velocity calculation both need changes in order to permute the elements of the data encoding, rather than simply changing their values. For our GA, we used the Order Crossover [13]. For PSO, we used IHDPSO's crossover. Their methodologies are quite similar. Each extracts a piece of an encoded candidate solution and uses it to start a new candidate solution. The remainder of the new solution, in both cases, is drawn from another candidate solution. Duplication is avoided (maintaining validity of the permutation) and order is preserved. Where our GA used a mutation operator that switched two genes, our PSO uses a more disruptive reverse operator that changes the ordering of a sequence. It's possible this is the chief reason for the difference in convergence speeds.

Overall, the genetic algorithm found its best solution, 652, over 100k generations. The particle swarm optimization best solution was not far behind, at 640. One major difference between the two algorithms was the generation when the best solution was found. The genetic algorithm found its optimum solution much earlier (around generation 20000) whereas the particle swarm optimization used almost all of the generations to find the best solution. As the rate of progress diminishes, we should mention our GA reached the 600 threshold in about 18,000 generations with a population of 20 chromosomes, while our PSO reached a fitness score of 600 after about 28,000 generations with a population of 100 particles.

In later generations, the genetic algorithm population appears to have converged (to high score solution) whereas the particle swarm optimization algorithm does not converge in our interpretation of the algorithm. Either way does not seem to affect the ability to find optimal solutions.

As for performance, the difference between the time taken by each algorithm to work through 100k generations was negligible. However, as the population sizes were radically different (PSO is larger by a factor of 10), it appears that the PSO operates much faster.

The most important aspect to each algorithm was to make sure randomness was injected into each population to maintain a dynamic search.

Conclusions

Our effort flavours PSO with GA crossover and multiple swarms/populations. We borrowed also from TSP solutions to render an inherently continuous problem solver capable of addressing this discrete challenge. But our program does little to further the discrete PSO (DPSO) cause. A very good solution was eventually found. But breaking the 600 threshold takes, on average, about 28,000 generations, which is slower than our GA.

Clearly, PSO strength lies in its ability to search a continuous solution space. Various efforts to contort the PSO algorithm into a discrete problem solver have met with moderate success. Earlier canonical PSO work on inertia weights and constriction factors remarks that 10,000 generations were more than ever required for a different problem [17]. This serves as some implicit condemnation of discrete PSO that requires significantly more iterations through the solution space.

Our algorithm showed little sensitivity to parameter adjustments. This may be partially attributable to an unclear presentation of the IHDPSO algorithm.

By adding DMSPSO to IHDPSO, we managed to undo one of PSO's advantages over GA. Our two source algorithms combine to create a large parameter set that rivals GA. The tidiness of PSO's minimalist parameter set is lost.

Thoughts for the future

We could perhaps have chosen a different model problem. TSP relevance to SGP is no greater than other discrete problems. N-queens solutions have as much in common (a methodology to permute valid candidate solutions) without TSP's edge-weighting [18]. A model based on flow shop scheduling problems, with the groups serving as the flow shop machines, might prove useful.

Other PSO variations offer some appeal. Perhaps the influence of a cadre of global best particles in some sort of Top N selection or amalgamation process would lead to effective operation. Leveraging various topologies could improve results.

Reflections

[Paul] PSO is a newer domain than GA, and consequently offers less volume of research. Both GA and PSO are better suited to continuous than discrete problems. In retrospect, I wish I had pursued an ACO solution. However, the assignment is without doubt an effective learning tool.

The algorithms we chose to model have not enjoyed mainstream success. Choosing relatively new papers has the advantage of offering unexplored possibilities. This is particularly valuable when the well-known work has not reached exalted levels. The IHDPSO is not particularly well written, and important pieces are unclear or omitted altogether. But the algorithm does appear to be inspired by some solid work. The DMSPSO has better credentials, and the paper is well written. Exploration into the benefits of multi-swarm architectures remains of interest, although it's not clear whether the isolation of the particles adds much of a benefit. The extra processing time could be spent on running a smaller population over more generations. It is beyond our scope here, but worth exploring the relative performance of these two approaches, where the total particles processed are equal.

Many papers on DPSO are heavy on math and light on explanation. While this certainly suits some readers, and is easier for some writers, examples and illustrations are clearer for me. This left some of the material a little further out of reach. And, as mentioned earlier, several papers rather poorly articulate their messages. In summary, it was a challenge finding highly useful reference material.

At the end of the day, our IHD-DMS-PSO feels more like a blunt instrument than an elegant tool. We discover a good result, but through brute force. This is a little disappointing.

But even a negative result is a result!

[Rajan] I enjoyed working this algorithm. The difficult part of the process was researching an adequate adaptation of the PSO to suit our needs. There are many papers written on the PSO but very few give detailed enough explanations to copy their implementation. Another snag was that we needed a TSP version of the PSO, which apparently is not its intended purpose.

Overall using both algorithms was a very intellectually stimulating experience.

References

[1] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: a tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, 1996.

[2] "COMP658 F13: Assignment 2 Questions." [Online]. Available: http://scis.lms.athabascau.ca/mod/forum/discuss.php?d=38038&parent=112134. [Accessed: 11-Nov-2013].

[3] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm Intell.*, vol. 1, no. 1, pp. 33–57, 2007.

[4] J. Kennedy and R. Eberhart, "Particle swarm optimization," in , *IEEE International Conference on Neural Networks, 1995. Proceedings*, 1995, vol. 4, pp. 1942–1948 vol.4.

[5] J. Kennedy and R. Mendes, "Population structure and particle swarm performance," in *Proceedings of the 2002 Congress on Evolutionary Computation, 2002. CEC '02*, 2002, vol. 2, pp. 1671–1676.

[6] T. Hendtlass, "Preserving Diversity in Particle Swarm Optimisation," in *Developments in Applied Artificial Intelligence*, P. W. H. Chung, C. Hinde, and M. Ali, Eds. Springer Berlin Heidelberg, 2003, pp. 31–40.

[7] W. Pang, K.-P. Wang, C.-G. Zhou, L.-J. Dong, M. Liu, H.-Y. Zhang, and J.-Y. Wang, "Modified particle swarm optimization based on space transformation for solving traveling salesman problem," in *Proceedings of 2004 International Conference on Machine Learning and Cybernetics, 2004*, 2004, vol. 4, pp. 2342–2346 vol.4.

[8] W.-N. Chen, J. Zhang, H. S.-H. Chung, W.-L. Zhong, W. Wu, and Y. Shi, "A Novel Set-Based Particle Swarm Optimization Method for Discrete Optimization Problems," *IEEE Trans. Evol. Comput.*, vol. 14, no. 2, pp. 278–300, 2010.

[9] K.-P. Wang, L. Huang, C.-G. Zhou, and W. Pang, "Particle swarm optimization for traveling salesman problem," in *2003 International Conference on Machine Learning and Cybernetics*, 2003, vol. 3, pp. 1583–1585 Vol.3.

[10] X. H. Shi, Y. Zhou, L. M. Wang, Q. X. Wang, and Y. C. Liang, "A DISCRETE PARTICLE SWARM OPTIMIZATION ALGORITHM FOR TRAVELLING SALESMAN PROBLEM," in *Computational Methods*, G. R. LIU, V. B. C. TAN, and X. HAN, Eds. Springer Netherlands, 2006, pp. 1063–1068.

[11] X. H. Shi, Y. C. Liang, H. P. Lee, C. Lu, and Q. X. Wang, "Particle swarm optimization-based algorithms for TSP and generalized TSP," *Inf. Process. Lett.*, vol. 103, no. 5, pp. 169–176, Aug. 2007.

[12] F. A. N. Huilian, "Discrete Particle Swarm Optimization for TSP based on Neighborhood [J]," *J. Comput. Inf. Syst.*, vol. 6, no. 10, pp. 3407–3414, 2010.

[13] J.-Y. Potvin, "Genetic algorithms for the traveling salesman problem," *Ann. Oper. Res.*, vol. 63, no. 3, pp. 337–370, 1996.

[14] J. J. Liang and P. N. Suganthan, "Dynamic multi-swarm particle swarm optimizer," in *Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005*, 2005, pp. 124–129.

[15] M. Lovbjerg, T. K. Rasmussen, and T. Krink, "Hybrid particle swarm optimiser with breeding and subpopulations," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2001, vol. 2001, pp. 469–476.

[16] M. Crepinsek, S.-H. Liu, and M. Mernik, "Exploration and exploitation in evolutionary algorithms: A survey," *ACM Comput Surv*, vol. 45, no. 3, pp. 35:1–35:33, Jul. 2013.

[17] R. C. Eberhart and Y. Shi, "Comparing inertia weights and constriction factors in particle swarm optimization," in *Proceedings of the 2000 Congress on Evolutionary Computation, 2000*, 2000, vol. 1, pp. 84–88 vol.1.

[18] X. Hu, R. C. Eberhart, and Y. Shi, “Swarm intelligence for permutation optimization: a case study of n-queens problem,” in *Proceedings of the 2003 IEEE Swarm Intelligence Symposium, 2003. SIS ’03*, 2003, pp. 243–246.